Metody obliczeniowe w nauce i technice - sprawozdanie 10

Łukasz Jezapkowicz

21.05.2020

1 Szukamy przybliżonego rozwiązania równania van der Pol'a:

$u''(t) + \mu u'(t)(u(t)^2 - 1) + u(t) = 0$

poprzez sprowadzenie równania do układu równań 1-ego stopnia z wprowadzeniem nowej zmiennej $v = u'(t)$:

$u' = v$

$v' = -u + \mu v(1 - u^2)$

Uruchomić programy rozwiązujące w/w równanie różniczkowe:

- Przeanalizować działanie programów, oszacować zbieżność rozwiązania.

- Narysować (np. za pomocą gnuplota) wykresy na podstawie danych wynikowych.

W sprawozdaniu opisać działanie metod użytych w w/w programach.

Na początku przeprowadzę analizę poszczególnych programów. Każdy z trzech załączonych programów posiada dwie wspólne, identyczne metody *func* oraz *jac*.
Zacznę więc od nich moją analizę.
W *GSL* by rozwiązywać jakąkolwiek metodą równanie różniczkowe należy zdefiniować t.zw. *ODE System* (Ordinary Differential Equations System). Definiujemy go przy pomocy zdefiniowanego typu *gsl_odeiv2_system*, którego konstruktor przyjmuje następujące parametry:
- *int* (∗ *function*) (*double t*, *const double y*[], *double dydt*[], *void* ∗ *params*)
- *int* (∗ *jacobian*) (*double t*, *const double y*[], *double* ∗ *dfdy*, *double dfdt*[], *void* ∗ *params*)
- *size_t dimension*
- *void* ∗ *params*

Parametr *dimension* określa ilość równań w naszym układzie równań zaś *params* jest wskaźnikiem na pewne dodatkowe atrybuty (przekazywane później do funkcji oraz jakobianu).
Pierwszym argumentem jest funkcja, której zadaniem jest obliczyć wartości $f_i(t, y, params)$, dla argumentów (t, y) i dodatkowych parametrów w *params*, i wpisać je do tablicy *dydt*. W przypadku sukcesu funkcja zwraca zdefiniowaną wartość *GSL_SUCCESS*, każda inna traktowana jest jako błąd.
Drugim argumentem jest funkcja, której zadaniem jest obliczyć wartości pochodnych cząstkowych $\frac{\partial f_i(t,y,params)}{\partial t}$ i wpisać je do tablicy *dfdt*. Powinna również wpisać wartości jakobianu J_{ij} w tablicy jednowymiarowej *dfdy*, dla której $J(i, j) = dfdy[i * dimension + j]$, gdzie *dimension* opisane zostało wcześniej. Funkcja powinna zwracać *GSL_SUCCESS* tak jak poprzednio.

Każdy z podanych trzech programów implementuje identyczne funkcje $func$ oraz $jacobian$, które zostaną przekazane jako dwa pierwsze argumenty do gsl_odeiv2_system. Spójrzmy więc na nie.

```
int
func (double t, const double y[], double f[],
      void *params)
{
  (void)(t); /* avoid unused parameter warning */
  double mu = *(double *)params;
  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
  return GSL_SUCCESS;
}
```

Funkcja jako parametr y przyjmuje tablice dwuelementową, dla której $y[0] = u$ oraz $y[1] = u' = v$. Odpowiednie wartości zostaną wpisane do tablicy f. W argumencie $params$ przekazujemy wartość μ. Funkcja kolejno ignoruje parametr t, rzutuje wartość μ, wpisuje wartości do tablicy f zgodnie z naszym dwuelementowym układem równań $u' = v$, $v' = -u + \mu v(1 - u^2)$ i zwraca wartość $GSL\_SUCCESS$.

```
int
jac (double t, const double y[], double *dfdy,
     double dfdt[], void *params)
{
  (void)(t); /* avoid unused parameter warning */
  double mu = *(double *)params;
  gsl_matrix_view dfdy_mat
    = gsl_matrix_view_array (dfdy, 2, 2);
  gsl_matrix * m = &dfdy_mat.matrix;
  gsl_matrix_set (m, 0, 0, 0.0);
  gsl_matrix_set (m, 0, 1, 1.0);
  gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
  gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
  dfdt[0] = 0.0;
  dfdt[1] = 0.0;
  return GSL_SUCCESS;
}
```

Funkcja przyjmuje identyczne parametry jak powyżej i wpisuje odpowiednie wartości do tablic $dfdy$ oraz $dfdt$. Funkcja kolejno ignoruje parametr t, rzutuje wartość μ, tworzy macierz $2x2$ związaną z $dfdy$ i wypełnia ją następująco: $\begin{bmatrix} 0.0 & 1.0 \\ -2.0 * \mu * u * v - 1.0 & -\mu * (u * u - 1.0) \end{bmatrix}$. Odpowiednie wartości wynikają z odpowiednich wartości pochodnych cząstkowych:
$\frac{df[0]}{u} = \frac{d(v)}{u} = 0.0$
$\frac{df[0]}{v} = \frac{d(v)}{v} = 1.0$
$\frac{df[1]}{u} = \frac{d(-u+\mu v(1-u^2))}{u} = -1.0 - 2.0 * \mu * v * u$
$\frac{df[1]}{v} = \frac{d(-u+\mu v(1-u^2))}{v} = \mu(1.0 - u^2)$.
Następnie funkcja wypełnia tablice $dfdt$ zerami ponieważ $f[0] = v$ oraz $f[1] = -u + \mu v(1 - u^2)$ nie zależą od t. Na koniec funkcja zwraca $GSL\_SUCCESS$.

Teraz możemy przejść do poszczególnych programów.

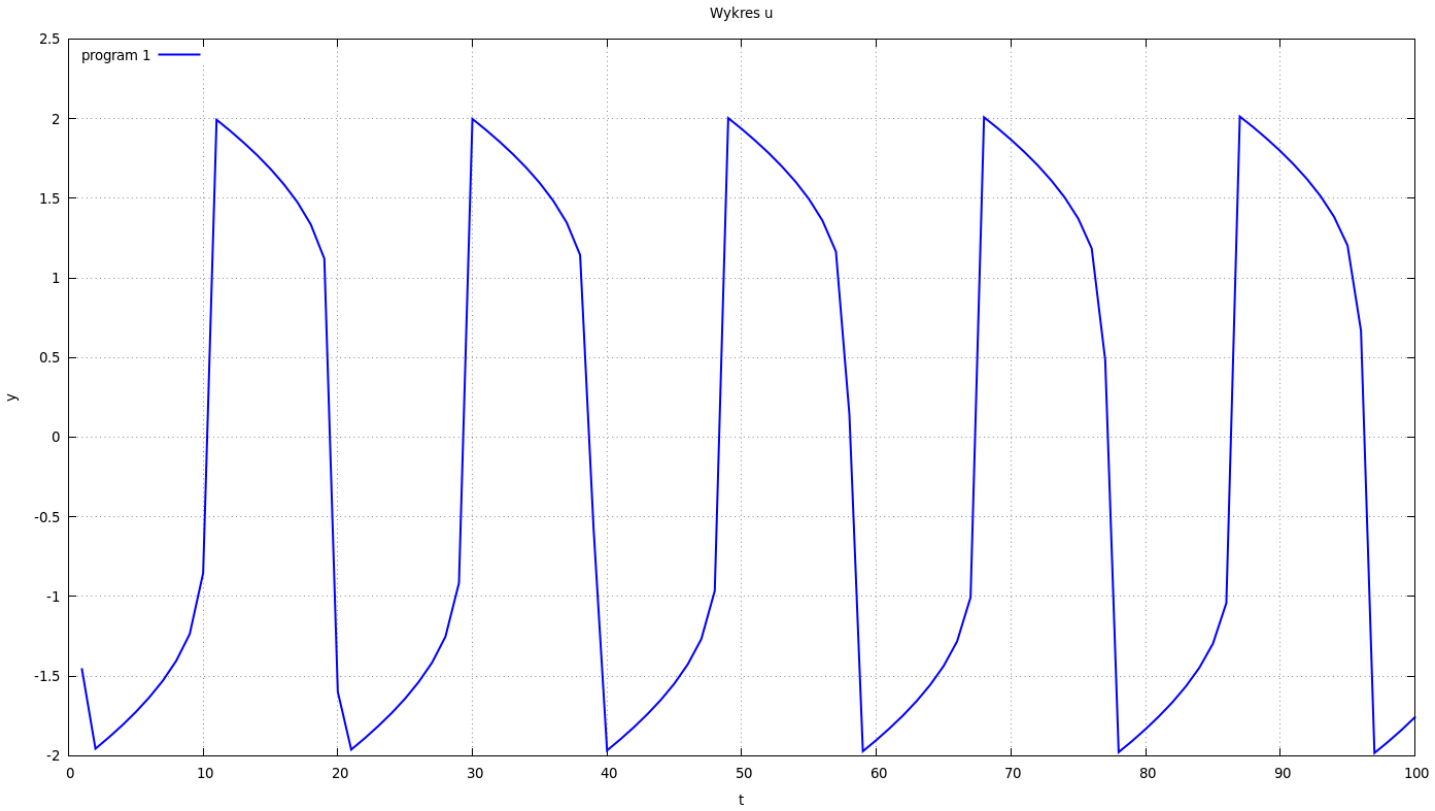
1.1 Program 1

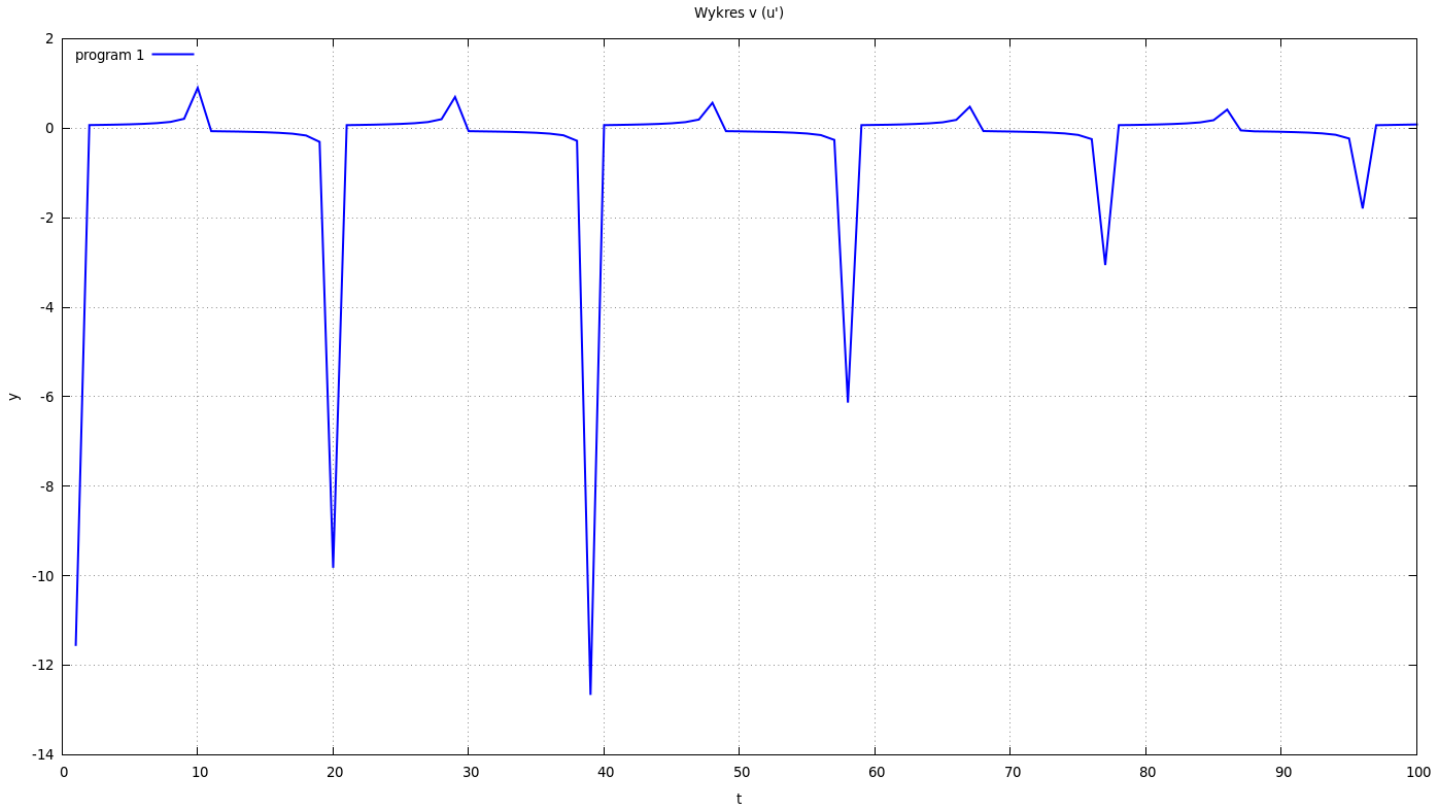
```
int
main (void)
{
  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};

  gsl_odeiv2_driver * d =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd,
                                   1e-6, 1e-6, 0.0);
  int i;
  double t = 0.0, t1 = 100.0;
  double y[2] = { 1.0, 0.0 };

  for (i = 1; i <= 100; i++)
    {
      double ti = i * t1 / 100.0;
      int status = gsl_odeiv2_driver_apply (d, &t, ti, y);

      if (status != GSL_SUCCESS)
        {
          printf ("error, return value=%d\n", status);
          break;
        }

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_driver_free (d);
  return 0;
}
```

Program nie przyjmuje żadnych parametrów. Wartość μ przyjmuje jako 10. Następnie tworzy opisany powyżej *gsl_odeiv2_system sys* dla opisanych wcześniej *func*, *jac*, rozmiaru układu równań równego 2 oraz dodatkowego parametru μ. Następnie tworzony jest obiekt typu *gsl_odeiv2_driver*, który pozwala na proste wykonywanie kolejnych kroków w rozwiązywaniu naszego układu równań. Jego konstruktor *gsl_odeiv2_driver_alloc_y_new* przyjmuje parametry:
- *sys* - zdefiniowane wcześniej
- *gsl_odeiv2_step_rk8pd* - rodzaj algorytmu wdrażającego kolejne kroki, tutaj jest to metoda $Runge - Kutta\ Prince\ Dormand(8,9)$
- $1e-6$ - rozmiar kroku (h)
- $1e-6$ - absolute error
- 0.0 - relative error

Absolute error oraz relative error to odpowiednio e_{abs} oraz e_{rel} w równaniu
$D_i = e_{abs} + e_{rel} * (a_y|y_i| + a_{dydt}h|y_i'|)$ wyrażającym pożądany poziom błędu D_i (a_y oraz a_{dydt} to pewne skalujące się czynniki związane z $y(t)$ oraz $y'(t)$).

Następnie wykonywane jest 100 kolejnych iteracji dla wartości początkowych t_i = 1.0 oraz u = 1.0, v = 0.0. W każdym kroku t_i zwiększane jest o 1.0, osiąga więc wartości $1.0, 2.0, ..., 100.0$. Kolejne kroki wykonywane są przy pomocy metody $gsl\_odeiv2\_driver\_apply$, która przyjmuje jako parametry zdefiniowany wcześniej gsl_odeiv2_driver, wartość t_i z poprzedniego kroku t, wartość t_i oraz wartości u oraz v w tablicy y. Metoda ta wykonuje kolejny krok z t do t_i i zwraca standardowo $GSL\_SUCCESS$ dla poprawnego wykonania kroku. Po każdym kroku program wypisuje wartości u oraz v. Na koniec program zwalnia pamięć przy pomocy metody $gsl\_odeiv2\_driver\_free$. Poniżej widoczne są wykresy wartości dla u oraz v:





Podstawowa metoda Rungego-Kutty jest prostą metodą iteracyjną pozwalającą rozwiązać równanie postaci $y' = f(x, y)$ dla znanej początkowej wartości $y(x_0) = y_0$. Wzory pozwalające obliczać kolejne wartości y dla wielkości kroku h są następujące:
$y_{n+1} = y_n + \Delta y_n$,
$\Delta y_n = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$,
gdzie,
$k_1 = hf(x_n, y_n)$,
$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1)$,
$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_2)$,
$k_4 = hf(x_n + h, y_n + k_3)$.
Metoda *Runge – Kutta Prince Dormand* działa na podobnej zasadzie ma jednak inny wzór na y_{n+1} oraz inną ilość zmiennych k_i oraz inne na nie wzory.
Metoda *Runge – Kutta Prince Dormand* jest oczywiście zbieżna. Przyjmując, że nasz rozmiar kroku nie będzie malał (a *GSL* dopuszcza taką możliwość) to wykonanych zostanie $\frac{100.0-0.0}{10^{-6}} = 10^8$ kroków. Ilość wykonanych kroków w dowolnym wywołaniu funkcji $*\_apply$ nie może przekroczyć pewnej zdefiniowanej wartości x. W najgorszym wypadku zostanie więc wykonanych $x*100$ kroków.

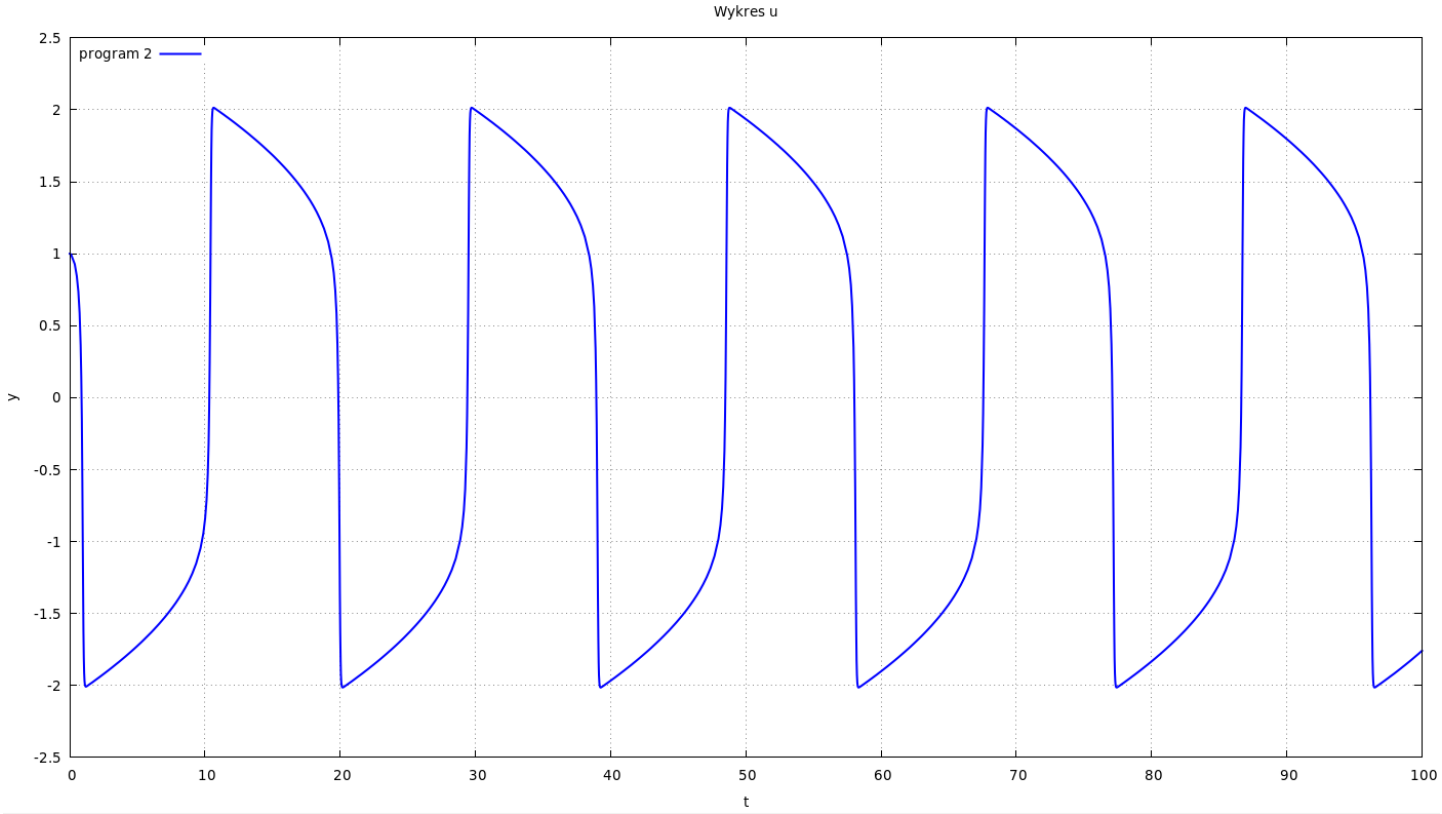
1.2 Program 2

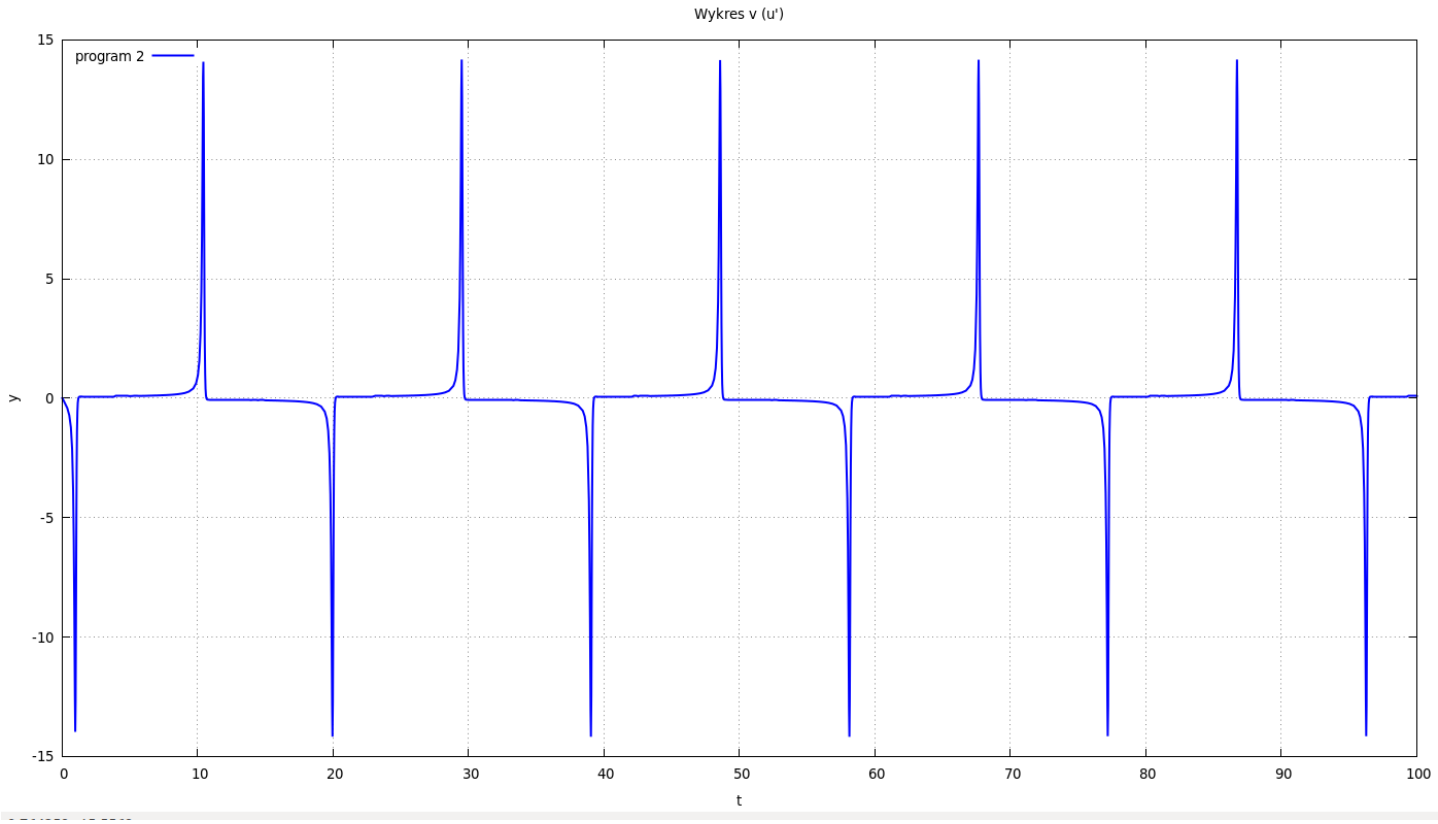
```c
int
main (void)
{
  const gsl_odeiv2_step_type * T
    = gsl_odeiv2_step_rk8pd;

  gsl_odeiv2_step * s
    = gsl_odeiv2_step_alloc (T, 2);
  gsl_odeiv2_control * c
    = gsl_odeiv2_control_y_new (1e-6, 0.0);
  gsl_odeiv2_evolve * e
    = gsl_odeiv2_evolve_alloc (2);

  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};

  double t = 0.0, t1 = 100.0;
  double h = 1e-6;
  double y[2] = { 1.0, 0.0 };

  while (t < t1)
    {
      int status = gsl_odeiv2_evolve_apply (e, c, s,
                                            &sys,
                                            &t, t1,
                                            &h, y);

      if (status != GSL_SUCCESS)
        break;

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_evolve_free (e);
  gsl_odeiv2_control_free (c);
  gsl_odeiv2_step_free (s);
  return 0;
```

Program nie przyjmuje żadnych parametrów. Wartość μ przyjmuje jako 10. Na początku tworzony jest obiekt typu *gsl_odeiv2_step_type* symbolizujący sposób rozwiązywania układu (algorytm wdrażający kolejne kroki). Tutaj przyjęty został *gsl_odeiv2_step_rk8pd* czyli tak jak poprzednio *Runge – Kutta Prince Dormand*$(8, 9)$. Następnie tworzone są kolejno obiekty:
- *gsl_odeiv2_step* czyli obiekt symbolizujący funkcję wykonującą kolejne kroki algorytmu, tutaj jako parametry przyjęty został wcześniej zdefiniowany *gsl_odeiv2_step_type* oraz rozmiar układu równań czyli 2.
- *gsl_odeiv2_control* czyli obiekt odpowiedzialny za monitorowanie błędów numerycznych w każdym kroku programu, tutaj jako parametry przyjmuje wcześniej opisane absolute error i relative error (o wartościach odpowiednio $1e-6$ oraz 0.0).
- *gsl_odeiv2_evolve* czyli obiekt, który łączy w sobie działanie dwóch wyżej wymienionych obiektów i pozwala na łatwe wykonywanie kolejnych kroków programu, tutaj jako parametr przyjęty jest rozmiar układu czyli 2.

Następnie program tworzy obiekt typu gsl_odeiv2_system, zmienne t, t_1 oraz tablice y o znaczeniu identycznym co w poprzednim programie. Dodatkowo tworzona jest zmienna $h = 1e-6$ symbolizująca wielkość kroku. Następnie w pętli wywoływana jest funkcja $gsl\_odeiv2\_evolve\_apply$, która wykonuje kolejny krok programu. Funkcja przyjmuje kolejno parametry:
- zdefiniowany wcześniej obiekt typu gsl_odeiv2_evolve
- zdefiniowany wcześniej obiekt typu $gsl\_odeiv2\_control$
- zdefiniowany wcześniej obiekt typu gsl_odeiv2_step
- zdefiniowany wcześniej obiekt typu gsl_odeiv2_system
- wartość t w danym kroku
- wartość graniczną t_1
- wielkość kroku czyli h
- tablicę wartości y

Funkcja po poprawnym wykonaniu zapisuje nowe wartości w zmiennych t oraz tablicy y. Po każdym wywołaniu wypisywane są wartości dla konkretnego kroku. Na końcu zwalniana jest pamięć przy pomocy odpowiednich funkcji. Poniżej widoczne są wykresy wartości dla u oraz v:





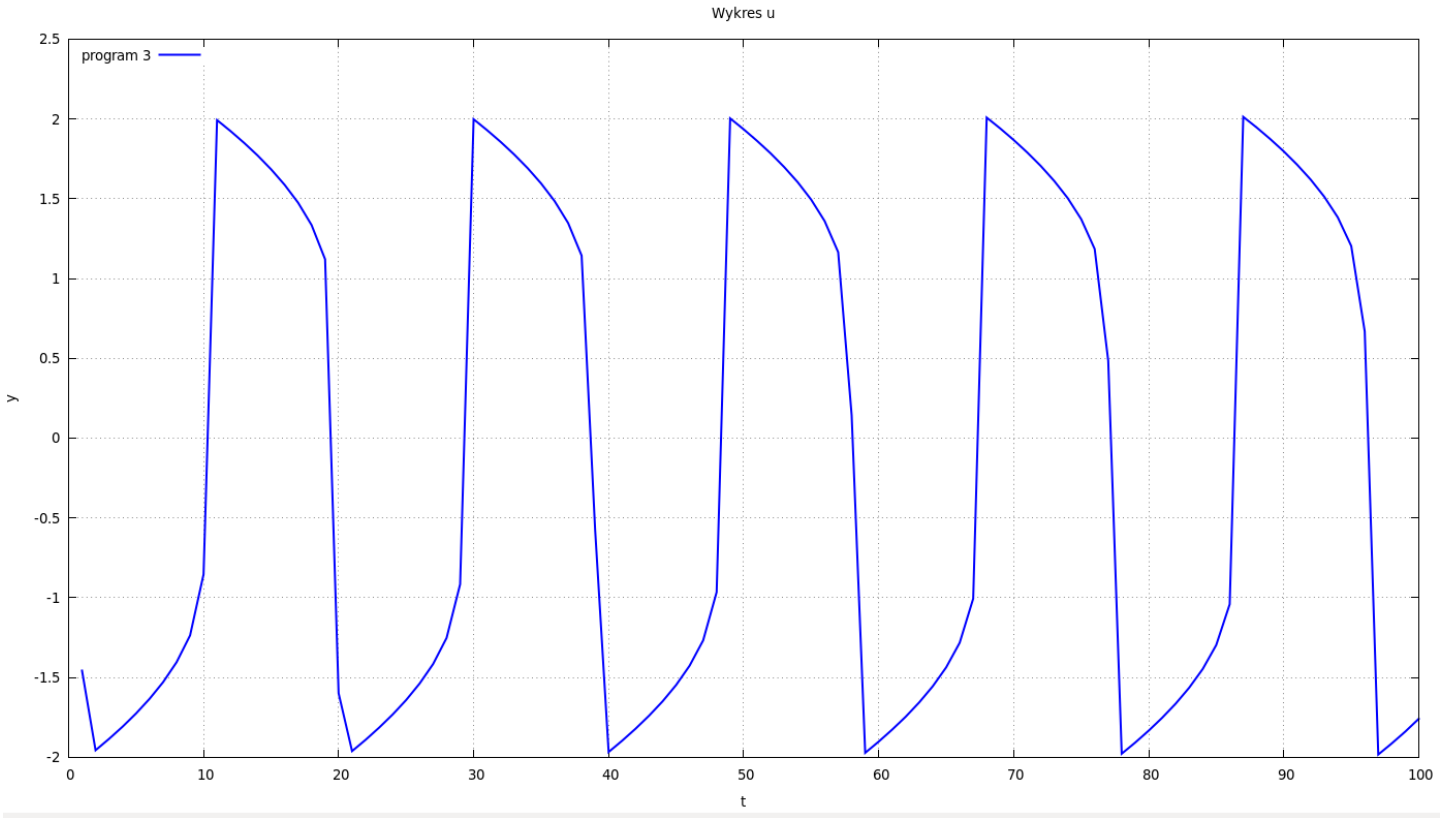
Metoda *Runge – Kutta Prince Dormand* jest oczywiście zbieżna. Przyjmując, że nasz rozmiar kroku nie będzie malał (a *GSL* dopuszcza taką możliwość) to wykonanych zostanie $\frac{100.0-0.0}{10^{-6}} = 10^8$ kroków. Ilość wykonanych kroków w dowolnym wywołaniu funkcji $*\_apply$ nie może przekroczyć pewnej zdefiniowanej wartości x. W najgorszym wypadku zostanie więc wykonanych $x * n$ kroków, gdzie n to ilość iteracji w pętli while.

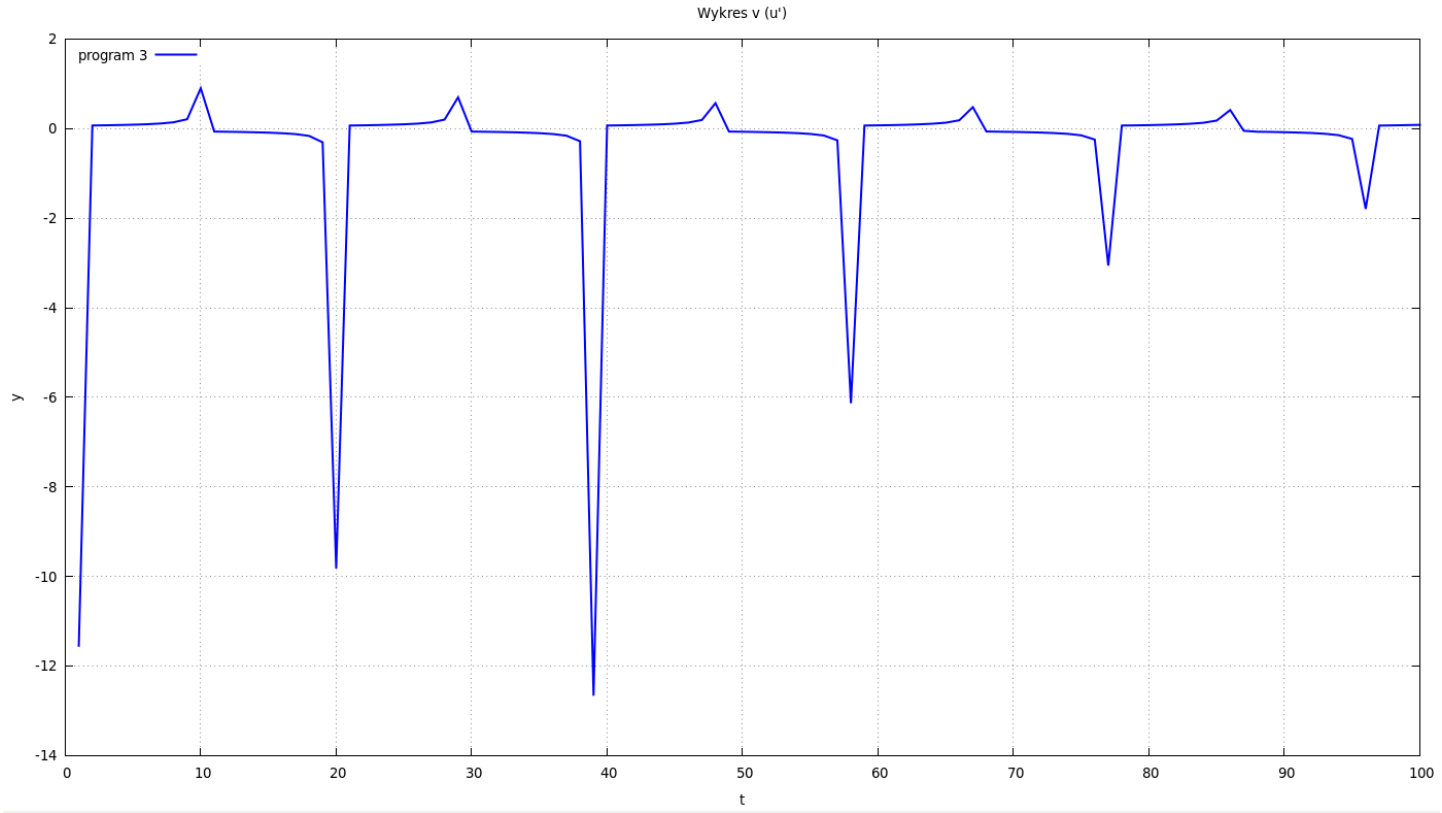
1.3 Program 3

```c
int
main (void)
{
  double mu = 10;
  gsl_odeiv2_system sys = { func, jac, 2, &mu };

  gsl_odeiv2_driver *d =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk4,
                                   1e-3, 1e-8, 1e-8);

  double t = 0.0;
  double y[2] = { 1.0, 0.0 };
  int i, s;

  for (i = 0; i < 100; i++)
    {
      s = gsl_odeiv2_driver_apply_fixed_step (d, &t, 1e-3, 1000, y);

      if (s != GSL_SUCCESS)
        {
          printf ("error: driver returned %d\n", s);
          break;
        }

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_driver_free (d);
  return s;
}
```

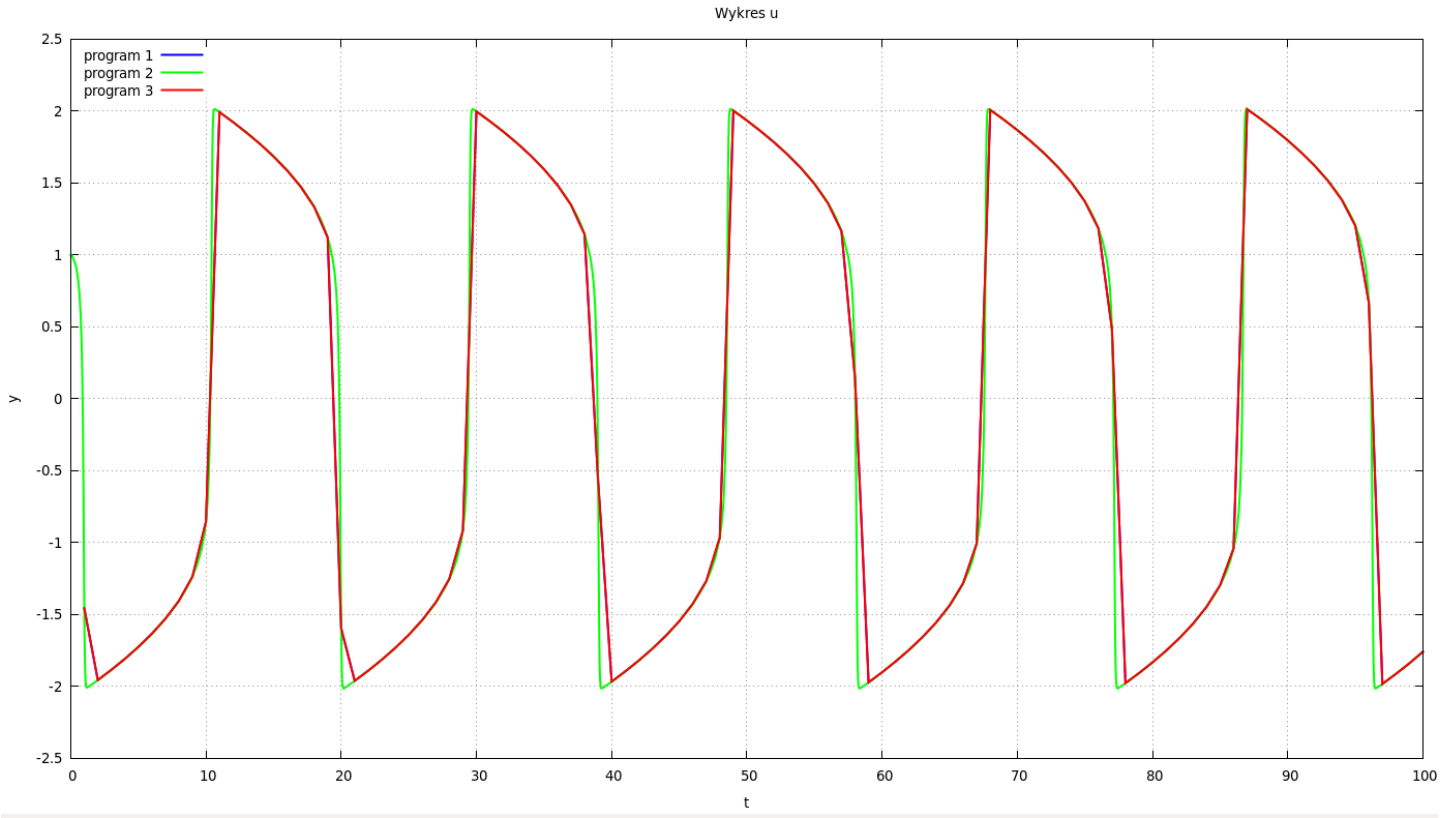
Program działa niemal symetrycznie do programu 1. Opiszę teraz jedyne znaczące róznice. W obiekcie typu gsl_odeiv2_driver przyjęta została inna metoda wdrażania kolejnych kroków - klasyczna metoda $Runge - Kutta$ ($gsl\_odeiv2\_step\_rk4$). Ponadto jako początkową wielkość kroku przyjęto tutaj $1e-3$ a absolute error i relative error są równe $1e-8$. Wszystkie początkowe wartości są takie same jak w programie 1. Następnie wykonywane jest 100 iteracji, w których wywoływana jest funkcja $gsl\_odeiv2\_driver\_apply\_fixed\_step$. Jej działanie polega na wykonaniu n kroków wielkości h. Jako kolejne parametry funkcja ta przyjmuje:
- zdefiniowany wcześniej obiekt typu gsl_odeiv2_driver
- wartość t w danym kroku
- wielkość kroku czyli h
- ilość kroków czyli n
- tablicę wartości y

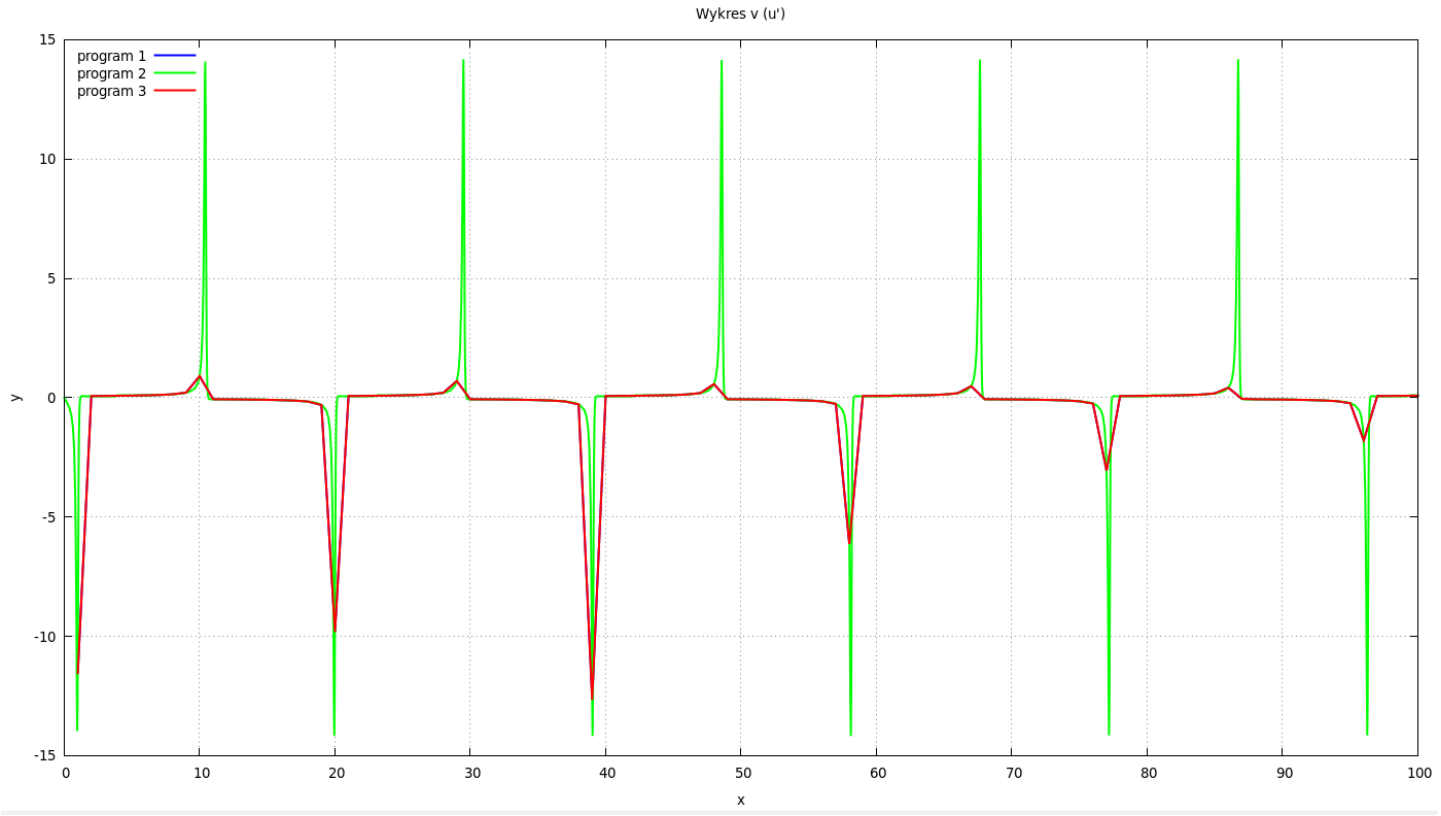
Ponieważ wykonywane jest 1000 kroków wielkości $1e-3$ więc w każdym kroku t zmienia się łącznie o 1.0 czyli identycznie jak w programie 1. W każdej iteracji wypisywane są wartości dla konkretnego kroku. Na końcu zwalniana jest pamięć przy pomocy odpowiedniej funkcji.
Metoda $Runge-Kutta$ jest oczywiście zbieżna. Przyjmując, że nasz rozmiar kroku nie będzie malał (a GSL dopuszcza taką możliwość) to wykonanych zostanie $\frac{100.0-0.0}{10^{-3}} = 10^5$ kroków. Ilość wykonanych kroków w dowolnym wywołaniu funkcji $*\_apply$ nie może przekroczyć pewnej zdefiniowanej wartości x. W najgorszym wypadku zostanie więc wykonanych $x * 100$ kroków. Poniżej widoczne są wykresy wartości dla u oraz v:


Wykres u
program 3
y
t
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
10
20
30
40
50
60
70
80
90
100


Wykres v (u')
program 3
y
t
2
0
-2
-4
-6
-8
-10
-12
-14
0
10
20
30
40
50
60
70
80
90
100

1.4 Wykresy wspólne





1.5 Wniosek

Jak widać wykres dla programu 1 dla u oraz v pokrył się z innym programem. Po sprawdzeniu okazuje się, że pokrył się z wykresem dla programu 3. Można więc powiedzieć, że programy okazały się równoważne. Program 1 oraz 3 obliczają pary (t, u, v) dla 100 elementów zaś program 2 dla ponad 700 elementów. Można więc wysunąć wniosek, że wykresy dla programu 2 okazały się najdokładniejsze, ponieważ zostało wykonanych najwięcej kroków. GSL udostępnia bardzo dużo różnych metod rozwiązywania kolejnych kroków, w tym sprawozdaniu pojawiły się dwie takie metody. Można więc dla wielu różnych metod i różnych parametrów takich jak długość skoku czy błąd bezwględny/względny wykonać odpowiednie obliczenia i sporządzić wykresy w celu znalezienia najlepszego odwzorowania (np. stosując taktykę "który wykres powtórzy się najwięcej razy"). Jak widać GSL jest bardzo dobrym rozwiązaniem do rozwiązywania układu równań różniczkowych i warto stosować go w obliczeniach numerycznych takich problemów.